Scoping in Standard ML

Luca Abeni, Csaba Kiraly

April 11, 2016

Bindings

- Per associare nomi a valori, abbiamo gia visto:
  - **val**: aggiunge binding all'ambiente (esempio: **val** x = 1;)
  - **val rec**: per funzioni ricorsive (rende il simbolo visibile all'espressione che ne stabilisce il valore)
  - **fun**: sintassi abbellita ("zucchero sintattico") per **val rec**
- Abbiamo visto esempi con scope globale

```
val x = 1;
val sq = x * x;
sq;
val x = 2;
sq;
```

- *Che si può dire di blocchi annidati (scope locale / non-locale)?*
  - *Scoping statico o dinamico?*

Scoping

```
val v = 1;
val x = 2;
val f = fn x => x + v;
```

- Visibilità di `x`?
  - **fn** crea un blocco di annidamento
  - crea anche nuovo binding (per il parametro formale) in tale blocco
  - può mascherare un binding esistente
  - smette di mascherare alla fine del blocco
- Visibilità di `v` all'interno di `f`?
  - simbolo non-locale (alcuni linguaggi funzionali dicono "free variable")
  - ereditato dal blocco esterno

Scoping / 2

- Scoping statico o dinamico?

```
val v = 1;
val f = fn x => x + v;
val v = 2;
f 3;
```

- Se scoping statico, `f 3` è valutato a `4`
- Se scoping dinamico, `f 3` è valutato a `5`

Modificare l'Ambiente Locale

- Parole chiave per introdurre binding nell'ambiente locale di un blocco
  - Binding non attivi fuori dal blocco
  - **let** declaration **in** expression **end**
  - **local** declaration1 **in** declaration2 **end**
- Modifica ambiente locale (fra **in** e **end**), no effetti fuori dal blocco
  - Binding locali al blocco
  - Attivati quando si entra nel blocco, disattivati quando si esce
- Notare differenze fra **let** e **local**
  - **local** vuole dichiarazione fra **in** e **end**
    - Modifica ambiente locale di tale dichiarazione
  - **let** vuole espressione fra **in** e **end**
    - Crea blocco di annidamento per tale espressione

Ambiente Locale: Esempio

- Esempio:
  - la funzione fattoriale dovrebbe essere definita solo su numeri naturali...
  - ...ma ML non conosce unsigned int!
  - Cosa succede con
    fun fact n = **if** n = 0 **then** 1 **else** n ∗ fact (n − 1) se invoco fact ~1???
  - Evitiamo il problema con un check:

```
val rec fact = fn n =>
  if n < 0 then ~1 else
    if n = 0 then 1 else n * fact(n - 1);
```

- Problema: **if** n < 0... è valutata ad ogni chiamata ricorsiva
  - Si può evitare usando **local**?

local fact

```
local
  val rec ffact = fn n =>
    if n = 0 then 1 else n * ffact(n - 1)
in
  val rec fact = fn n =>
    if n < 0 then ~1 else ffact (n)
end;
```

- Problema precedente (check su `n < 0` ad ogni chiamata ricorsiva) risolto controllando `n` prima di invocare la funzione ricorsiva
- **local** usato per "nascondere" la funzione "fattoriale senza check" (`ffact`)
- *Esercizio: implementare fattoriale con check usando* **let**
- *Nota la differenza:* **let** *definisce binding locale in un'espressione*

let fact

```
val rec fact = fn n =>
  let
    val rec ffact = fn n =>
      if n = 0 then 1 else n * ffact(n - 1)
  in
      if n < 0 then ~1 else ffact (n)
  end;
```

- *Esercizio: implementare la versione tail recursive di fact con local o let*

local tail recursive fact

```
local
  val rec fact1 = fn n => fn res =>
    if n = 0 then res else fact1 (n - 1) (n * res)
in
  val rec fact = fn n =>
    fact1 n 1
end;
```